# 第十七章 多线程

**本节所讲内容：**

**17.1 python之线程介绍**

**17.2 python之多线程编程**

**17.3 Threading**

## 17.1 多线程介绍

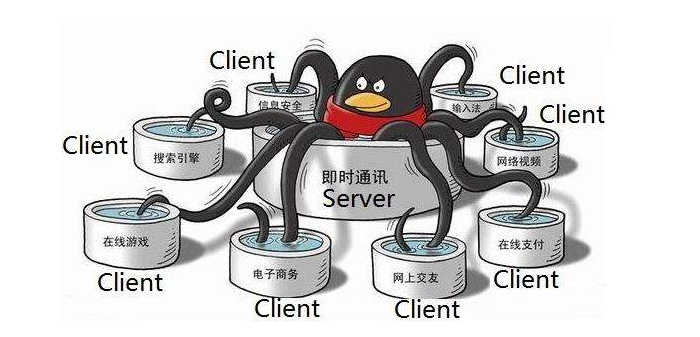
### 17.1.1 进程

**计算机程序只不过是磁盘中可执行的二进制（或其它类型）的数据。它们只有在被读取到内存中，被操作系统调用的时候才开始它们的生命期。**

**进程是程序的一次执行。每个进程都有自己的地址空间，内存，数据栈以及其它记录其运行轨迹的辅助数据。操作系统管理在其上运行的所有进程，并为这些进程公平地分配时间。**

### 17.1.2 线程

**线程（有时被称为轻量级进程）跟进程有些相似，不同的是，所有的线程运行在同一个进程中，共享相同的运行环境。我们可以想像成是在主进程或“主线程”中并行运行的“迷你进程”。**



### 17.1.3 进程和线程的关系

**进程中第一个线程是主线程，主线程可以创建其他线程；**

**其他线程也可以创建线程，线程之间是平等的；**

**进程有父进程和子进程，独立的内存空间，它的唯一标识符：pid。**

### 17.1.4 进程和线程功能

**进程：能够完成多任务，比如一台电脑上能同时运行多个QQ；**

**线程：能够完成多任务，比如一个QQ中的多个聊天窗口；**

### 17.1.5 进程和线程的区别和联系

**1. 一个程序至少有一个进程,一个进程至少有一个线程（这个线程是主线程）。**

**2. 线程的划分尺度小于进程(资源比进程少)，使得多线程程序的并发性高。**

**3. 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。**

**4. 线程不能够独立执行，必须依存在进程中。**

**5. 线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.**

**6. 进程是系统进行资源分配和调度的一个独立单位，进程是资源分配的单位，线程是cpu调度的单位。**

### 17.1.6 进程与线程的优缺点

**多进程的优点是稳定性好，一个子进程崩溃了，不会影响主进程以及其余进程。但是缺点是创建进程的代价非常大，因为操作系统要给每个进程分配固定的资源，并且，操作系统对进程的总数会有一定的限制，若进程过多，操作系统调度都会存在问题。**

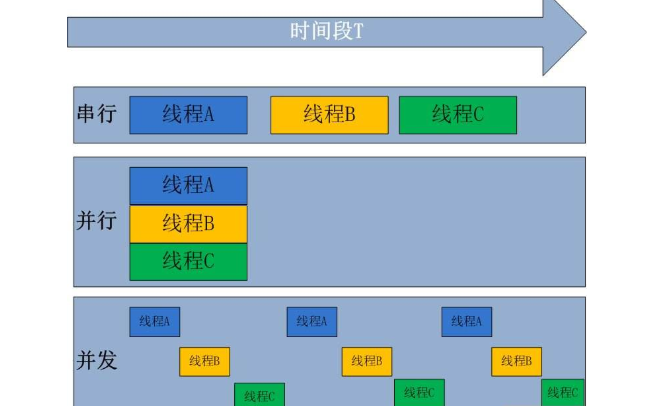
**多线程优点是效率较高一些，但是致命的缺点是任何一个线程崩溃都可能造成整个进程的崩溃，因为它们共享了进程的内存资源池。**

### 扩展：

### 什么叫并发，什么叫并行？

**并发：在一个时间段，处理多个任务，单核也可以并发（CPU分时间片）；**

**并行：在同一个时刻，处理多个任务，必须多核才能并行；**



### 什么是GIL？

**GIL为全局解释器锁，GIL本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。**

**某个线程想要执行，必须先拿到GIL，我们可以把它看做一个线程的‘通行证’。每个进程中只有一个GIL。**

**17.2 \_thread方法**

**Python中使用线程有两种方式：函数或者是类来包装线程对象**

**函数式：调用\_thread模块中的start\_new\_thread()方法来产生一个新的线程  
import \_thread  
import time  
*# 为线程定义一个函数*def print\_time( threadName, delay):  
 count = 0  
 while count < 5:  
 time.sleep(delay)  
 count += 1  
 print ("%s: %s" % ( threadName, time.ctime(time.time()) ))  
*# 创建两个线程  
#但是这个模块我们不推荐，因为底层封装的时候它的主线程不会等待子线程的结束(非阻塞)！  
#官方以及我们推荐再封装模块Threading，所以在这里大家了解下有这个\_thread方法就好*try:  
 \_thread.start\_new\_thread( print\_time,("Thread-1", 2, ) )  
 \_thread.start\_new\_thread( print\_time,("Thread-2", 4, ) )  
except:  
 print ("Error: 无法启动线程")  
while True:  
 pass**

**运行结果如下：**

**Thread-1: Wed Jan 16 11:53:03 2019**

**Thread-2: Wed Jan 16 11:53:05 2019**

**Thread-1: Wed Jan 16 11:53:05 2019**

**注意：**

**这个模块我们不推荐，因为底层封装的时候它的主线程不会等待子线程的结束！官方以及我们推荐再封装Threading，所以在这里大家了解下。**

## 17.3 Threading

**当一个进程启动之后，会默认产生一个主线程，因为线程是程序执行流的最小单元，当设置多线程时，主线程会创建多个子线程！**

**联系进程的知识点，主线程会等待所有的子线程执行完成程序再退出**

### 17.3.1 threading介绍

**1、threading是对thread模块的再封装，线程并发运行并且共享内存；**

**2、threading模块支持守护线程；**

**3、守护线程：**

**守护线程会在"该进程内所有非守护线程全部都运行完毕后,守护线程才会挂掉"。并不是主线程运行完毕后守护线程挂掉。**

**守护线程守护的是：当前进程内所有的子线程**

**主线程在其他非守护线程运行完毕后才算运行完毕（守护线程在此时就被回收）。因为主线程的结束意味着进程的结束，进程整体的资源都将被回收，而进程必须保证非守护线程都运行完毕后才能结束。**

### 17.3.2 threading常用方法

**1、 threading.currentThread(): 返回当前的线程变量。**

**注：threading.currentThread().name 返回当前线程的名字。**

**2、 threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。**

**3、 threading.activeCount():返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。**

**4、 构造方法：   
Thread(group=None, target=None, name=None, args=(), kwargs={})**

**group: 线程组，目前还没有实现，库引用中提示必须是None；   
target: 要执行的方法；   
name: 线程名；   
args/kwargs: 要传入方法的参数。**

**5、 实例方法：   
isAlive(): 返回线程是否在运行。正在运行指启动后、终止前。   
getName/setName(name)：获取/设置线程名。**

**start()：线程准备就绪，等待CPU调度。  
join([timeout])：阻塞当前环境中的子线程，直到调用此方法的线程终止或到达指定的timeout（可选参数）。**

### 17.3.3 threading案例

**import threading  
import time  
def say(name):  
 print('first start time %s'%time.ctime())  
 print('你好,%s'%name)  
 time.sleep(3)  
 print('first stop time %s '%time.ctime())  
def hello(name):  
 print('second start time %s'%time.ctime())  
 print('你好,%s'%name)  
 time.sleep(3)  
 print('second stop time %s '%time.ctime())  
def main():  
 print('我是main')  
*#threading.current\_thread().name代表的是这个线程的name*print('\_\_\_主线程开始\_\_\_',threading.current\_thread().name)  
*#target 调用的是函数对象  
#args 是函数对象传递的实参*t1 = threading.Thread(target=say,args=('for',))  
t2 = threading.Thread(target=hello,args=('while',))  
t3 = threading.Thread(target=main,args=())  
*#设置线程的名字*t1.setName('越努力，越幸福')  
*#启动线程*t1.start()  
*#打印线程的名字*print(t1.getName())  
t2.start()  
t3.start()  
print('\_\_\_主线程结束\_\_\_',threading.current\_thread().name)**

**运行结果如下：**

**\_\_\_主线程开始\_\_\_ MainThread**

**first start time Mon Feb 18 14:16:43 2019**

**越努力，越幸福**

**你好,for**

**second start time Mon Feb 18 14:16:43 2019**

**你好,while**

**我是main**

**\_\_\_主线程结束\_\_\_ MainThread**

**first stop time Mon Feb 18 14:16:46 2019**

**second stop time Mon Feb 18 14:16:46 2019**

### 17.3.4 守护线程

**无论是进程还是线程，都遵循：守护xxx会等待主xxx运行完毕后被销毁**

**需要强调的是：运行完毕并非终止运行**

**#1.对主进程来说，运行完毕指的是主进程代码运行完毕**

**#2.对主线程来说，运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕，主线程才算运行完毕**

**详细解释：**

**#1 主进程在其代码结束后就已经算运行完毕了（守护进程在此时就被回收）,然后主进程会一直等非守护的子进程都运行完毕后回收子进程的资源，才会结束，**

**#2 主线程在其他非守护线程运行完毕后才算运行完毕（守护线程在此时就被回收）。因为主线程的结束意味着进程的结束，进程整体的资源都将被回收，而进程必须保证非守护线程都运行完毕后才能结束**

**1. 一个子线程**

**一个子线程，并且设置为守护线程的时候，这个守护线程会在主线程运行完毕后强制关闭。**

**import threading  
import time  
def say(name):  
 print('%s is start ' % name)  
 time.sleep(3)  
 print('%s is stop'%name)  
print('\_\_\_主线程开始\_\_\_')  
*#主线程会等待子线程结束才结束，相当于主线程是一个守护线程*t = threading.Thread(target=say,args=('for',))  
*#设置守护线程，并且这个守护线程会在主线程完毕后自动挂掉!  
#注意该守护线程要在start之前设置*t.setDaemon(True)  
*#开启子线程*t.start()  
print('\_\_\_主线程结束\_\_\_')**

**运行结果如下：**

**\_\_\_主线程开始\_\_\_**

**for is start**

**\_\_\_主线程结束\_\_\_**

**2. 多个子线程**

**当有多个子线程时，守护线程就会等待所有的子线程包括主线程运行完毕后，守护线程才会挂掉，在子线程或者主线程运行完毕之后，守护线程如果没有执行完，也会停止运行。**

**import threading,time  
def say(name):  
 print('%s is start ' % name)  
 time.sleep(4)  
 print('%s is stop'% name)  
def hello(name):  
 print('%s is start ' % name)  
 time.sleep(3)  
 print('%s is stop'%name)  
print('\_\_\_主线程开始\_\_\_')  
t1 = threading.Thread(target=say,args=('for',))  
t2 = threading.Thread(target=hello,args=('while',))  
*#设置t1为守护线程*t1.setDaemon(True)  
*#启动线程*t1.start()  
t2.start()  
print('\_\_\_主线程结束\_\_\_')**

**运行结果如下：**

**\_\_\_主线程开始\_\_\_**

**for is start**

**while is start**

**\_\_\_主线程结束\_\_\_**

**while is stop**

### 17.3.5 列表式使用

**利用列表的方式，实现我们多线程并发执行的效果.**

**案例1：**

**import threading  
import time  
def run(number):  
 print('%s…… starting '%number)  
 time.sleep(2)  
 print('-------------')  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('\_\_\_主线程开始\_\_\_',threading.current\_thread().name)  
 thread\_list=[]  
 for i in range(1,5):  
 #创建，实例化一个线程  
 t = threading.Thread(target= run,args=(i,))  
 thread\_list.append(t)  
 #将实例化的线程都放到列表中，统一执行  
 for t in thread\_list:  
 t.start()  
 print('\_\_\_主线程结束\_\_\_',threading.current\_thread().name)**

**运行结果如下：**

**\_\_\_主线程开始\_\_\_ MainThread**

**1…… starting**

**2…… starting**

**3…… starting**

**4…… starting**

**\_\_\_主线程结束\_\_\_ MainThread**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**17.3.6 join的用法**

**join的功能是完成线程同步，即主线程任务结束后进入阻塞状态，等待子线程任务全部结束后再终止。**

**案例：**  
**import threading  
import time  
def run(number):  
 print('%s…… starting '%number)  
 time.sleep(2)  
 print('\_\_\_\_\_\_\_\_\_\_\_\_')  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('\_\_\_主线程开始\_\_\_',threading.current\_thread().name)  
 thread\_list=[]  
 for i in range(1,5):  
 *#创建，实例化一个线程* t = threading.Thread(target= run,args=(i,))  
 thread\_list.append(t)  
 *#将实例化的线程都放到列表中，统一执行* for t in thread\_list:  
 t.start()  
 t.join()  
 print('\_\_\_主线程结束\_\_\_',threading.current\_thread().name)**

**运行结果如下：**

**\_\_\_主线程开始\_\_\_ MainThread**

**1…… starting**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**2…… starting**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**3…… starting**

**\_\_\_\_\_\_\_\_\_\_\_\_**

**4…… starting**

**\_\_\_\_\_\_\_\_\_\_\_\_**

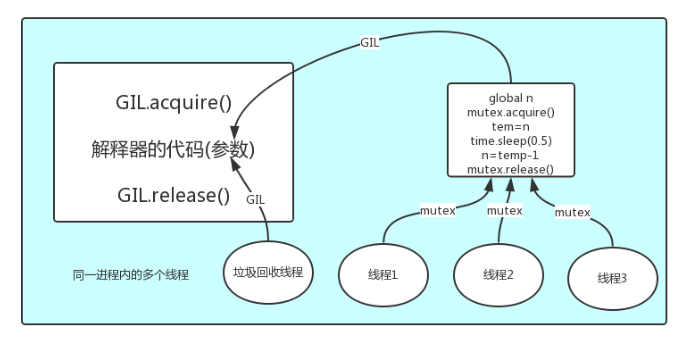
**\_\_\_主线程结束\_\_\_ MainThread**

**分析：join阻塞了主线程，前面主线程结束后，继续执行子线程，而我们加入join后，主线程就会等待所有子线程全部执行完毕，才能正常运行。**

### 17.3.7 线程锁

**线程锁是为了保证共享数据的一致性，线程锁的意义在与同一个时间内，多个线程同时修改共享数据，如果不加锁的话，会造成数据的不一致，当我们加上锁以后，保证在同一个时间里，只能运行一个线程修改共享数据，其他线程阻塞等待该线程的结束。**

**注意：结束后一定要释放锁，不然会一直阻塞下去**



**面向对象线程：  
import threading,time  
class MyThread(threading.Thread):  
 *#自带的一个实例方法运行* def run(self):  
 global num  
 time.sleep(1)  
 num +=1  
 msg = self.name + ' set num to '+ str(num)  
 print(msg)  
num = 0  
def test():  
 for i in range(5):  
 t = MyThread()  
 t.start()  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()**

**运行结果如下：**

**Thread-1 set num to 1**

**Thread-5 set num to 2**

**Thread-4 set num to 3**

**Thread-3 set num to 4**

**Thread-2 set num to 5**

**上锁的操作：**

**import threading,time  
num = 0  
*#加一个锁*lock = threading.Lock()  
class MyThread(threading.Thread):  
 def run(self):  
 lock.acquire()  
 *#s声明全局* global num  
 *#沉睡一秒* time.sleep(1)  
 *#num = num +1* num += 1  
 *#self.name 是线程的名字* msg = self.name + ' set num to '+ str(num)  
 print(msg)  
 lock.release()  
def test():  
 *#开启线程* for i in range(5):  
 *#创建一个实例* t = MyThread()  
 t.start()  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()**

**运行结果如下：**

**Thread-1set num to 1**

**Thread-4set num to 2**

**Thread-3set num to 3**

**Thread-2set num to 4**

**Thread-5set num to 5**

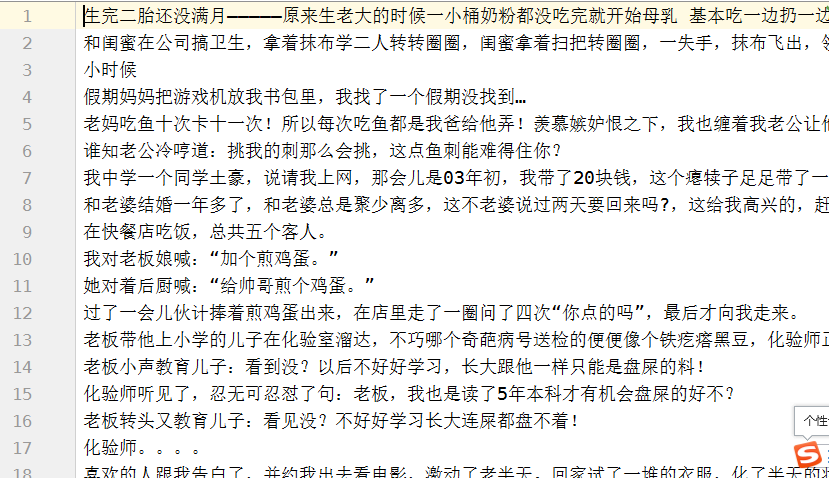
**实战：多线程爬取糗事百科**

**import requests  
from lxml import etree  
import threading,time  
lock = threading.Lock()  
class MyThread(threading.Thread):  
 base\_url = 'https://www.qiushibaike.com/text/page/{}'  
 headers = {  
 'Referer': 'https://www.qiushibaike.com/text/page/2/',  
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.75 Safari/537.36'  
 }  
 def \_\_init\_\_(self,page):  
 super(MyThread, self).\_\_init\_\_()  
 self.base\_url = self.base\_url.format(page)  
 *#自带的一个实例方法运行* def run(self):  
 lock.acquire()***#得到锁*

*#网络请求* **content = requests.get(self.base\_url,headers=self.headers).content.decode()**

*#把前端网页变成xpath可匹配得到的文本，然后利用xpath匹配文本* **xpath\_content = etree.HTML(content).xpath('//\*[@class="content"]/span/text()')  
 print(xpath\_content)  
 with open('1.txt','a',encoding='utf-8') as file:  
 for i in xpath\_content:  
 file.write(i.strip()+'\n')  
 lock.release()  
def main():  
 for i in range(1,4):  
 t = MyThread(i)  
 t.start()  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()**

**运行结果如下：**



**总结：**

**17.1 python之线程介绍**

**17.2 python之多线程编程**

**17.3 Threading**